AI-Human Centaur With a Ghost: Human–AI–Virtual Voevodsky Metacognition for Foundations-First Research in Category Theory and Homotopy Type Theory (HoTT)

Douglas C. Youvan

doug@youvan.com

August 29, 2025

This paper proposes a working protocol for Human–AI–Virtual Voevodsky (HAVV) metacognition: a collaboration model where a human sets intent and constraints, an AI executes and refactors rapidly, and a standing "virtual Voevodsky" enforces foundations-first rigor. The aim is to convert enticing but informal "emergence" into certified mathematics. Our protocol climbs an epistemic ladder—heuristic → concrete → certified—using explicit gates. Heuristics are allowed only as provisional guides; the concrete phase instantiates claims in a fully specified model (for example, finite sets and total functions); the certified phase elevates the result to a parametric theorem in a proof assistant, where equalities are proofs and universal properties replace diagrams. As a first step, we treat a Yoneda witness: given rep: RA ≃ A, define ϕ_X: Hom(X, RA) → Hom(X, A) by v ↦ (rep ∘ v) and prove ϕ_X is bijective and natural in X. We show how this moves from probe intuition to exact verification and then to a general, mechanized theorem, with artefact provenance and obligation closure as primary metrics. The outcome is a reusable pattern: engines emit proof obligations, not annotations; views are extracted from proofs, not the other way around.

Keywords: human–AI collaboration, metacognition, HAVV, category theory, Yoneda, representability, naturality, universal properties, univalent foundations, proof assistants, Coq, certified computation, obligation generator, GenesisBoard, centaur mathematics.

46 pages

A collaboration with GPT-5 – Thinking

Dedicated to my mentors from 1965 – 1975: Dr. Robert Reed (The Manhattan Project, born Ft. Scott, Kansas) and Don and Denis Buche (Vietnam Veterans, born Arma, Kansas).

1. Executive summary

1.1 Definition of HAVV (Human–AI–Virtual Voevodsky) metacognition

Human–AI–Virtual Voevodsky (HAVV) metacognition is a collaboration protocol with three explicit roles and a set of gated behaviors that move ideas from intuition to certified mathematics. The **human** sets intent, scope, and stop conditions, and decides what counts as success. The **AI** executes quickly, refactors artifacts, and maintains traceability across representations (text, code, proofs). The **virtual Voevodsky** is a standing foundational critic that rejects informal equalities and demands that all claims be expressed as universal properties and proven equalities. The protocol climbs an epistemic ladder with three stages: **heuristic** (exploratory signals and conjectures), **concrete** (fully specified models where statements can be checked as literal truths), and **certified** (parametric theorems inside a proof assistant). Progression is controlled by **gates**: we leave the heuristic stage when ambiguity persists, we enter the concrete stage when semantics can be made exact, and we only claim generality after a proof term is produced. In HAVV, views are extracted from proofs, not the other way around; and "equal" means there exists a proof object witnessing the equality.

1.2 Problem: heuristic "emergence" without proofs

Emergent patterns discovered by probe engines and growth processes are compelling, but their evidentiary status is weak when supported only by counters, visual motifs, or "declared equalities." A diagram with dashed arrows does not certify a property like representability or naturality. Without an explicit model and proof obligations, it is easy to overfit to proxies (for example, triangle counts or pullback-square counters) that do not entail a universal property. In our motivating case, a probe showed robust product growth while never producing a certified Yoneda witness; the counters were consistent with interesting structure, yet they did not establish the required bijections ϕ_X: Hom(X, RA) → Hom(X, A) or the equation $\phi_Y(u \circ f) = \phi_X(u) \circ f$. The broader problem is common: research gets stuck at the level of plausible pictures. HAVV addresses this by replacing pictures with **semantics first**, then replacing semantics with **proofs**. Informal equalities become obligations; obligations produce either proofs or minimal counterexamples.

1.3 Contribution: a repeatable protocol that moves ideas from heuristic to concrete to certified, with explicit roles and gates

This paper contributes a **repeatable protocol** and a **first worked example**. The protocol specifies roles (human, AI, virtual Voevodsky), gates (when to move up the ladder), and artifacts (models, obligations, proofs, and recorded provenance). The worked example treats a Yoneda witness: assuming rep: RA ≃ A, define $\phi_X(u) = \text{rep} \circ u$ from Hom(X, RA) to Hom(X, A) and prove ϕ_X is bijective and natural in X. We demonstrate the ladder in action: heuristic intuition from a probe,

an exact construction in finite sets and total functions, and a **parametric Coq development** where equalities are witnessed by proof terms. The deliverables are (a) a metacognitive protocol others can adopt, (b) an interface, "YonedaWitness," capturing the data and laws engines must satisfy, and (c) a pathway for engines to emit **proof obligations** rather than annotations. The value is practical: the same workflow can certify products, pullbacks, and representability, turning informal "emergence" into mathematics that survives scrutiny and composes with future results.

2. Roles and responsibilities

2.1 Human: intent, constraints, and stop conditions

The human leads with problem framing and accountable choices. Responsibilities:

- Set intent: the research question, success criteria, and the target property to certify (for example, "establish a natural bijection ϕ_X: Hom(X, RA) → Hom(X, A)").
- Declare constraints: admissible models, banned shortcuts, and resource limits (time, complexity, library surface).
- Specify stop conditions: what evidence is sufficient to pause, pivot, or conclude (for example, "produce a proof term for naturality, or exhibit a concrete counterexample triple (X, u, f)").
- Approve gates: authorize movement from heuristic to concrete to certified phases.
- Resolve tradeoffs: decide when to weaken a claim, sharpen assumptions, or defer a lemma.
- Own narrative: articulate the why of each step and decide which artifacts become part of the scientific record.

Outputs:

- A short specification stating the property, the intended models, and acceptance tests.
- A decision log recording gate approvals, deferrals, and rationale.

Common anti-patterns:

- Accepting visual plausibility as proof.
- Expanding scope without revising success criteria.
- Treating failures as noise rather than extracting counterexamples.

2.2 AI: construction, refactoring, and traceability

The AI is the high-throughput engineer of artifacts across representations. Responsibilities:

- Rapid construction: produce candidate models, code, and proofs consistent with the human's constraints.
- Refactoring: reorganize specifications, proofs, and code to reduce complexity and improve reuse.
- Obligation generation: turn claims into explicit obligations (equalities, universal properties) with clear inputs and outputs.
- Counterexample extraction: when proofs fail, mine minimal counterexamples from concrete models.
- Provenance and reproducibility: maintain hashes, dependency lists, and a clear chain from claims to proof objects or counterexamples.
- Documentation: produce plain-language summaries and formal statements with identical meaning.

Guarantees and limits:

- Guarantees: every equality in the certified phase is witnessed by a proof term; every diagram rendered is derivable from proofs.
- Limits: the AI does not mint axioms, conflate models, or hide failed goals; it reports open obligations explicitly.

Failure modes:

- Silent assumption creep (for example, accidental set-theoretic commitments).
- Proxy overreach (for example, relying on counters that do not entail a universal property).
- Overproduction of artifacts without a stable core specification.

2.3 Virtual Voevodsky: foundational audit and veto power

The virtual Voevodsky is a standing critic embodying foundations-first discipline. Responsibilities:

- Enforce "equality as proof": every claimed equality must be accompanied by an explicit witness derived from laws.

- Demand universal properties: products, pullbacks, and representability must be stated and verified via their universal characterizations.
- Guard generality: distinguish instance-level truths from parametric theorems; block claims of generality without proof terms.
- Prefer univalent formulations when appropriate: treat isomorphisms as transportable equalities in univalent settings and ensure statements are phrased to admit such transport.
- Police provenance: ensure that every rendered view is extractable from certified artifacts; forbid diagrams that cannot be justified.

Authority:

- Veto unproven claims advancing up the epistemic ladder.
- Require either a completed proof term or a minimal counterexample before scope expansion.

Success indicators:

- Theorems stated at the appropriate level of generality with explicit hypotheses.
- Proof obligations minimized by factoring common lemmas and interfaces (for example, a reusable YonedaWitness interface).
- A clean separation between exploratory heuristics and certified results, with traceable transitions between them.

3. The epistemic ladder

3.1 Heuristic phase: exploratory patterns and provisional signals

Purpose. Generate hypotheses cheaply while avoiding commitments that will be hard to unwind later. In this phase we tolerate proxy signals (for example, triangle counts, pullback-square counters, frequency of specific rewrite rules) but we treat them as prompts for questions, not evidence for claims.

Activities.

- Observe: run engines, record qualitative motifs and quantitative counters.
- Hypothesize: propose candidate invariants and target properties. Example: “there exists RA with a map rep: RA → A such that for each X, the function ϕ_X defined by precomposition is a bijection Hom(X, RA) → Hom(X, A).”
- Triage: list assumptions that would have to be true for the hypothesis to make sense (for example, identities and composition are total, hom-sets are well-defined, domains and codomains are typed).
- Design probes: specify what concrete measurements could quickly falsify the hypothesis.

Deliverables.

- A short hypothesis note: target property, proxies to watch, and quick falsifiers.
- A risk register: known gaps (for example, “no guarantee that RA exists in this growth regime”).

Exit gate “H→C” (Heuristic to Concrete).

- Trigger: ambiguity persists after a bounded amount of probing, or the same pattern appears across independent runs.
- Requirements: nominate at least one fully specified model where the property can be checked as literal truth or falsehood (for example, finite sets and total functions).
- Veto conditions: if the property is not even well-typed in the intended model, stop and revise the hypothesis.

3.2 Concrete phase: exact semantics in a fully specified model

Purpose. Replace suggestive pictures with exact objects and functions such that statements are decidable and counterexamples are computable.

Activities.

- Choose a model with total semantics. Example: the category of finite sets and total functions. In this model, arrows are functions, composition is function composition, equality of arrows is pointwise equality.
- Re-express the target property. Example: for each finite set X, evaluate the function ϕ_X: Hom(X, RA) → Hom(X, A) defined by v ↦ rep ∘ v and test bijectivity by counting and

comparing function tables; test naturality by checking that for each f: Y → X and u: X → RA we have rep ∘ (u ∘ f) = (rep ∘ u) ∘ f.

- Generate obligations from claims. Obligations are equalities or unique existence statements whose truth can be decided in the chosen model.
- Extract counterexamples when obligations fail. For bijection failure, produce a specific X and functions u or v that witness non-injectivity or non-surjectivity.

Deliverables.

- A concrete model file (for example, “objects = {X, Y, A, RA}, morphisms = explicit tables”).
- An obligations log with pass or fail per obligation and, if fail, a minimal counterexample triple (X, u, f).
- A provenance record (inputs, seeds, hashes, and transformation steps).

Exit gate “C→Ce” (Concrete to Certified).

- Trigger: obligations pass in at least one nontrivial concrete model and the property appears to be structural rather than accidental.
- Requirements: restate the property at the level of a general abstract interface (for example, a record for categories with objects, hom-sets, identity, composition, and laws), then attempt a proof that does not depend on model-specific features beyond the stated hypotheses.
- Veto conditions: if the property depends on model-specific quirks (for example, finiteness or cardinalities), either elevate those to explicit hypotheses or refuse generalization.

3.3 Certified phase: parametric theorems and proof objects

Purpose. Establish the claim once and for all in a proof assistant under clearly stated hypotheses, producing reusable theorems and interfaces.

Activities.

- Formalize the ambient structure. Define a category interface: objects, hom-sets, identity, composition, and laws (associativity, left and right identity).
- State the hypotheses precisely. Example: an isomorphism rep_iso: RA ≃ A consisting of to: RA → A and from: A → RA with equations to ∘ from = 1[A] and from ∘ to = 1[RA].

- Prove the target result parametrically. Example: for every object X, define ϕ_X(u) = to ∘ u and ψ_X(v) = from ∘ v; prove ψ_X(ϕ_X(u)) = u and ϕ_X(ψ_X(v)) = v for all u and v; prove naturality ϕ_Y(u ∘ f) = ϕ_X(u) ∘ f for all f: Y → X.
- Package the result as an interface. Example: a record YonedaWitness with fields {RA, rep_iso, ϕ, bijectivity, naturality}.
- Enforce view discipline. Any diagram or visualization must be derivable from the proof terms; no unproven adornments.

Deliverables.

- A compiled proof artifact (for example, a .vo file) and the source file containing definitions and theorems.
- Machine-checked statements of theorems and a list of hypotheses.
- A proof dependency graph (lemmas and their uses).

Exit gate “Ce→Deploy” (Certified to Deployment in workflows).

- Trigger: theorems are stable, hypotheses match the intended engines, and interfaces are minimal.
- Requirements: translate engine outputs into obligation sets that target the certified interfaces; instrument the engine to emit obligations and, optionally, proof terms.
- Veto conditions: if the certified result’s hypotheses are systematically unsatisfiable by engine outputs, either adjust the engine or weaken or refactor the target theorem.

3.4 Gate criteria, rollback rules, and invariants

Gate criteria (summary).

- H→C: at least one fully specified model exists and the hypothesis is well-typed in it; commit to obligations that decide the hypothesis in that model.
- C→Ce: obligations pass in a nontrivial instance and the claim can be stated abstractly without model-specific crutches; commit to an abstract interface and proof attempt.
- Ce→Deploy: the abstract interface aligns with what engines can produce; commit to obligation generation and visualization discipline.

Rollback rules.

- If an obligation fails in the concrete phase, roll back to the heuristic phase only to revise the specification, not to reweight proxies. Produce a minimal counterexample and annotate exactly which assumption failed.
- If the certified proof stalls, attempt to minimize the theorem (remove unnecessary generality) before enlarging the library or adding axioms. If minimization still fails, return to the concrete phase to search for counterexamples that refine the statement.
- If deployment reveals a mismatch between engine outputs and hypotheses, prefer modifying the engine to satisfy the certified interface; only then consider weakening the theorem.

Invariants across phases.

- Typing invariants: every arrow has defined domain and codomain; composition is only attempted when codomain and domain match.
- Equality invariants: an equality claim is either backed by a proof term (certified phase) or by literal equality in the chosen model (concrete phase); informal equalities are never accepted as evidence.
- Provenance invariants: every artifact (image, table, theorem) carries a pointer to its generating source (logs, code, proof term), and any reader can trace the chain from view to proof.

4. Protocol for HAVV sessions (10 steps)

4.1 Hypothesize target property and invariants

State a precise, falsifiable target in the language of the intended domain, along with invariants that must hold for the claim to be meaningful. Example: “Target property: there exists an object RA and an arrow rep: RA → A such that for every object X, the map ϕ_X: Hom(X, RA) → Hom(X, A), v ↦ (rep ∘ v), is a bijection natural in X.” Invariants: identities exist, composition is total and associative, hom-sets are well formed, and typing is enforced (domains and codomains match). Deliverable: a one-paragraph hypothesis and a bullet list of invariants. Anti-pattern: defining the target only by diagrams or counters; make the claim text-first and type-checkable.

4.2 Declare admissible models and banned shortcuts

Fix which concrete models are allowed and which shortcuts are forbidden. Admissible models must have exact semantics (for example, finite sets and total functions). Banned shortcuts

include “declared equalities” without witnesses, counting-based proxies as evidence, and hidden partial functions. Deliverable: a model contract stating what constitutes an object, an arrow, equality of arrows, and composition. Anti-pattern: silently switching between models (for example, mixing a graph rewrite system with set-theoretic reasoning) without translation rules.

4.3 Build a minimal concrete model with exact semantics

Construct the smallest nontrivial instance that can decide the target property. All objects and arrows are explicit, equality is literal in the model (for example, function equality), and all compositions are computed. Deliverable: a machine-readable artifact (for example, a JSON or table) listing objects, hom-sets, and composition tables. Guardrail: keep the instance minimal to reduce cognitive load and speed counterexample extraction.

4.4 Define obligations as universal properties and equalities

Translate the target property into a finite set of obligations. For a Yoneda witness: for each object X, (1) ϕ_X is injective, (2) ϕ_X is surjective, and (3) for all $f: Y \to X$ and all $u: X \to RA$, $\phi_Y(u \circ f) = \phi_X(u) \circ f$. If products or pullbacks are claimed, state their universal properties with uniqueness clauses. Deliverable: an obligations list with precise inputs and expected outputs. Anti-pattern: “look right” criteria or ad-hoc thresholds.

4.5 Attempt a general proof and record dependencies

Lift the property to a parametric statement in a proof assistant. Define an abstract interface (for example, a Category record with objects, hom-sets, id, composition, and laws) and state hypotheses (for example, an isomorphism rep_iso: $RA \simeq A$). Attempt the proof that produces reusable lemmas. Deliverable: a source file and a dependency list (which lemmas and axioms were used). Guardrail: if you need an axiom, stop and justify it; prefer isolating axioms behind minimal interfaces.

4.6 If proof fails, extract minimal counterexamples

Return to the concrete model to find the smallest witness of failure. For bijection failure, provide X and a pair of distinct v1, v2 in Hom(X, RA) with $\phi_X(v1) = \phi_X(v2)$ (non-injectivity), or provide v in Hom(X, A) with no preimage (non-surjectivity). For naturality failure, provide $f: Y \to X$ and $u: X \to RA$ with $rep \circ (u \circ f) \neq (rep \circ u) \circ f$. Deliverable: a counterexample triple with exact tables. Guardrail: prefer the smallest domain that witnesses failure.

4.7 Refactor the specification before refactoring proofs

When failure occurs, first adjust the statement or hypotheses rather than hacking the proof. Typical moves: add missing typing constraints, make previously implicit assumptions explicit, or weaken the claim to the precise strength warranted by evidence. Only after the statement is

stabilized should proof refactoring begin. Deliverable: a revised specification and a changelog explaining the delta. Anti-pattern: proof golfing around an incorrect statement.

4.8 Package success as an interface with laws and proof terms

When the proof succeeds, encapsulate it as a reusable interface with minimal fields and explicit laws. Example: define a record YonedaWitness with fields RA, rep_iso, and a family ϕ_X, together with proofs: for every X, ϕ_X has a two-sided inverse and satisfies naturality. Deliverable: an interface file and a short “how to satisfy” note for engine implementers. Guardrail: keep interfaces law-centric and avoid leaking implementation details.

4.9 Automate obligation generation from engine outputs

Bridge the engine to the certified layer by compiling engine states into obligation sets. The translator should: (1) read a graph snapshot, (2) construct a typed signature of objects and arrows, (3) detect candidate structures (for example, products, pullbacks, representability), and (4) emit obligations that target the certified interfaces. Deliverables: a schema for obligations and a log that pairs each view or claim with the obligations and proofs that justify it. Guardrail: all diagrams must be derivable from the proof objects produced.

4.10 Archive artifacts with hashes and provenance

For every session, produce a bundle with fixed identifiers: hypothesis note, model contract, concrete instance, obligations and results, proofs or counterexamples, and any rendered views. Include cryptographic hashes of source files and proof objects, toolchain versions, and compile commands. Deliverable: a reproducibility bundle that lets a third party rebuild the result from scratch. Guardrail: forbid untracked edits and orphaned images; every view must have a provenance path to its generating proof or counterexample.

5. Artefact taxonomy and provenance

5.1 Pictures, models, proofs: three classes with different evidentiary status

We separate artefacts into three classes with explicit evidentiary semantics.

(1) Pictures (views). These are human-facing renderings: graphs, commutative squares, growth timelines, and dashboards. A picture is never evidence by itself. A picture must be derivable from certified data. Rule P1: every picture must be generated by a view function that takes as input either a proof object or a counterexample witness. Rule P2: pictures are read-only; interactive toggles may change layout but cannot create claims. Rule P3: every picture carries a provenance footer with a short identifier that resolves to the exact proof or counterexample that justifies each drawn feature.

(2) Models (semantics). These instantiate a claim in a fully specified domain with exact evaluation rules. In the concrete phase, a model contains explicit objects, morphisms, and composition tables; equality of arrows is literal model equality (for example, function equality). Rule M1: models are the only place where we compute truth values outside a proof assistant. Rule M2: every model file includes totality checks for composition and typing checks for domains and codomains. Rule M3: counterexamples extracted from models must be minimal with respect to a predefined size measure (for example, smallest domain cardinality or shortest arrow list).

(3) Proofs (certificates). These are machine-checked derivations in a proof assistant under stated hypotheses. Rule C1: any equality stated in the certified phase must be backed by a proof term inhabiting the corresponding identity type. Rule C2: any universal property must be expressed as an existence-and-uniqueness statement, not as a drawing. Rule C3: every theorem records its hypotheses explicitly; downstream artefacts must include a map from their assumptions to these hypotheses.

Relation between classes. A picture is a lossy rendering of a proof or counterexample; a model is an executable semantics used to falsify or support conjectures; a proof is the only artefact that can legitimately support a claim of generality. Movement from model to proof requires restating results parametrically and discharging all model-specific assumptions or elevating them to explicit hypotheses.

5.2 Naming conventions, versioning, and hash-based lineage

Identifiers. Each artefact receives a stable identifier of the form PREFIX/YYYYMMDD/NNN, where PREFIX ∈ {PIC, MOD, PRF, OBL, CTX}, the date is UTC, and NNN is a zero-padded sequence index for that day. Examples: PRF/20250829/017 for a proof object; MOD/20250829/012 for a model instance; PIC/20250829/044 for a figure generated from PRF/20250829/017.

Hashes. We compute a cryptographic hash for every artefact over its canonical form.

- Pictures: hash the SVG or PDF bytes after stripping non-semantic metadata.
- Models: hash the JSON with keys sorted and whitespace normalized.
- Proofs: hash the proof assistant's compiled object (for example, .vo) and the normalized source.
- Obligation logs: hash the sorted list of obligations and results.
- Context files: hash the toolchain manifest and environment report.

Lineage. Each artefact declares its parents by identifier and hash, forming a directed acyclic graph.

- A picture lists the proof object that justifies it and the view function version.
- A model lists the hypothesis note and model contract it instantiates.
- A proof lists the interface files and lemmas it depends on, pinned by hash.
- An obligation log lists the engine snapshot that generated it and the certified interface it targets.

Versioning. Artefacts are immutable. Corrections are new artefacts with new identifiers that point back via a “supersedes” relation. We never replace an artefact in place; we deprecate it with an explicit reason code: ERR-SPEC (specification error), ERR-PROOF (proof bug), ERR-TOOL (toolchain defect), or VIEW-ONLY (visual correction without semantic change).

Resolution. A resolver script accepts an artefact identifier and returns the transitive closure of its lineage graph, along with hashes and toolchain versions. A paper figure cite is thus stable: PIC/20250829/044 resolves to PRF/20250829/017 and VIEWFN/1.3.2 with exact hashes.

5.3 Reproducibility bundles: source, obligations, proof terms, rendered views

Bundle structure. For each session or result, we publish a bundle with the following directories:

- 00_context: toolchain manifest (compilers, library versions), environment report, and random seed log if applicable.
- 10_spec: the hypothesis note, model contract, and interface definitions (for example, Category, Iso, YonedaWitness).
- 20_model: concrete instances (JSON), along with validation logs and any counterexamples produced.
- 30_obligations: machine-readable obligation sets, results, and minimal counterexample witnesses when a check fails.
- 40_proofs: proof sources and compiled objects, dependency list with hashes, and a machine-generated theorem index.
- 50_views: pictures generated from proofs or counterexamples, each with a provenance footer linking to parent hashes.

Build scripts. The bundle includes scripts with deterministic entry points:

- build_model: generate the concrete instance from a specification and seed; validate typing and totality.
- check_obligations: evaluate obligations on the model; emit pass/fail logs and counterexamples.
- build_proofs: compile the proof sources to objects; fail if the environment hash does not match 00_context.
- render_views: re-generate all figures from the current proof objects and view functions; fail if any view is not justified by a listed parent proof or counterexample.

Minimal guarantees. A third party with the same toolchain must be able to:

- Rebuild all proofs from source and verify object hashes match.
- Re-run obligation checks on the concrete model and reproduce pass/fail results and counterexamples.
- Re-render all figures and obtain identical hashes for the outputs.

Deviation reporting. If a rebuild yields different hashes, the scripts emit a diff summary indicating which stage diverged (model generation, obligation evaluation, proof compilation, or rendering) and which inputs differ (source text, toolchain version, environment variables).

Publication note. The paper cites bundle identifiers rather than embedding figures without provenance. In archival repositories, we deposit both the human-readable paper and the machine-verifiable bundle, ensuring that every claim in the text has a resolvable pointer to its generating artefacts.

6. Equality as proof, not annotation

6.1 Replace “declared equalities” with constructed equalities

HAVV treats every equality as something that must be **derived** from laws or universal properties, not drawn or asserted informally. In a category, equal arrows are justified by equations built from associativity and identity, or by uniqueness clauses. Example (Yoneda witness): let rep_iso: RA ≃ A with to: RA → A and from: A → RA. Define for any X: ϕ_X(u) = to ∘ u. If we also define uY = uX ∘ f : Y → RA and vX = to ∘ uX : X → A, then
vY = to ∘ uY = to ∘ (uX ∘ f) = (to ∘ uX) ∘ f = vX ∘ f
by associativity. The proof is the chain of equalities; no dashed arrow is accepted. In our

certified development this appears as a proof term inhabiting the identity type for Hom(Y, A) between ϕ_Y(uX ∘ f) and ϕ_X(uX) ∘ f.

Operational rule E1: do not “name and assert” equalities; **define** terms and **derive** their equalities.
Operational rule E2: any equality shown in a figure must correspond to a derivation that exists as a proof term (certified phase) or as literal equality in the chosen model (concrete phase).

6.2 Universal properties as uniqueness proofs rather than diagrams

Products, pullbacks, and representability are verified by **universal characterizations**, not by shapes. For a binary product P of A and B with projections π1: P → A and π2: P → B, the key law is: for any X and maps f: X → A, g: X → B, there exists a unique u: X → P with π1 ∘ u = f and π2 ∘ u = g. Consequence: to show u = u', it suffices to show π1 ∘ u = π1 ∘ u' and π2 ∘ u = π2 ∘ u'. For a pullback of f: A → C and g: B → C, uniqueness in the universal square plays the same role. For representability, a witness rep: RA → A supplies, for each X, a natural bijection Hom(X, RA) ≃ Hom(X, A). In all cases, **uniqueness turns families of commuting equalities into a single equality of mediating arrows**.

Operational rule U1: encode each purported structure as data plus a uniqueness clause.
Operational rule U2: prove equations between mediating arrows by projecting to their universal components and appealing to uniqueness.

6.3 Iso versus identity, and univalence-aware formulations

In classical category theory and in Coq without univalence, we work with an **isomorphism** RA ≃ A given by to and from with to ∘ from = 1[A] and from ∘ to = 1[RA]. Our Yoneda results are stated under this hypothesis and do not require any additional axioms. In a **univalent** setting, an isomorphism corresponds to a **path** RA = A, allowing transport along equalities rather than along isos. The HAVV protocol is compatible with both:

Design rule I1 (portable core): formulate theorems in terms of an Iso record so they compile in standard Coq.
Design rule I2 (univalence layer): when univalence is available, add lemmas that convert Iso RA A into a path RA = A and restate results in path form. This makes equalities computational and allows transporting structures across RA and A transparently.

Pragmatics. In concrete models like finite sets, we avoid meta-level axioms by checking function equality **extensionally** via tables. In abstract developments, we avoid reliance on functional extensionality by staying categorical: equalities are between arrows inside Hom, derived from associativity, identities, and universal properties, or witnessed by the Iso equations.

7. Instrumentation and metrics for HAVV

7.1 Proof density and obligation closure rate

Proof density measures how much certified content is produced relative to narrative or code. Let L_thm be the number of non-comment lines in proof sources and L_doc the number of lines in the prose that state or explain those theorems. Define density D = L_thm / (L_doc + 1). Target ranges depend on audience; for a research note we want D between 0.5 and 2.0, indicating that claims are backed by substantial formal content without overwhelming exposition. Obligation closure rate tracks how many obligations are discharged per session. If O_total is the number of obligations generated and O_closed the number proved, define R_close = O_closed / (O_total + 1). A healthy session pushes R_close upward while keeping O_total stable by refactoring common subgoals into lemmas.

Operational practice: display D and R_close in the session dashboard; regressions (drop in D or R_close) trigger a checkpoint discussion about scope and specification quality.

7.2 Counterexample latency and minimal counterexample size

Counterexample latency is the time between an obligation's creation and the discovery of a falsifying witness in a concrete model. Let t_emit be the timestamp when an obligation is emitted, t_cex when a counterexample is logged; define latency T_cex = t_cex − t_emit. Shorter T_cex indicates that the concrete model and the extractor are well aligned with the specification. Minimal counterexample size quantifies how small the witness can be made under a fixed size measure S (for example, number of objects, domain cardinality, or arrow count). We log the tuple (obligation id, T_cex, S_min). A good extractor not only finds counterexamples quickly but also minimizes S_min, which improves interpretability.

Operational practice: if T_cex is large or S_min bloats, revisit the model contract and extractor heuristics; ensure that obligations are stated in a way that enables incremental search and early stopping.

7.3 Refactor cost and proof reuse ratio

Refactor cost measures how expensive it is to evolve the specification without breaking the certified layer. Suppose a change set modifies K interfaces or lemmas. Let O_broken be the number of obligations or theorems that fail to compile after the change. Define refactor cost C_ref = O_broken / (K + 1). A low C_ref indicates that abstractions are well chosen and that common lemmas shield downstream proofs. Proof reuse ratio quantifies how much of the old proof graph survives a change. If V_old is the set of proof objects before and V_new after, define reuse U = |V_old ∩ V_new| / (|V_old| + 1). High U signals stable core lemmas and modularization.

Operational practice: when C_ref spikes or U drops, prefer strengthening interfaces (fewer fields, more laws) rather than scattering ad-hoc lemmas. Schedule periodic “library gardening” to merge duplicative proofs and expose missing abstractions.

7.4 Provenance completeness and view justification rate

Provenance completeness is the fraction of artefacts that carry a resolvable lineage to models or proofs. If A_total is the number of artefacts produced in a session and A_traced those with valid parents and hashes, define P_comp = A_traced / (A_total + 1). View justification rate is the fraction of rendered pictures whose features are generated exclusively from proofs or counterexamples. If V_total is the number of views and V_just the number that pass an automated check (no unbacked edges or labels), define J_view = V_just / (V_total + 1).

Operational practice: enforce a pre-publication gate requiring P_comp close to 1.0 and J_view close to 1.0. Any manual diagram editing must be flagged and either eliminated or accompanied by a script that reproduces the diagram from certified inputs.

8. Case study: Yoneda witness pipeline (first fruitful step)

8.1 Heuristic: a probe suggested “Yoneda in soup,” but only products grew

We began with an engine that rewrites graphs using rules resembling composition, pullbacks, and products. Heuristic counters (triangles, pullback-square counts, a boolean “yoneda”) suggested phases of growth. In practice, with the seed we examined, the engine fired only product-pair rules. The graph expanded with objects that looked like products and projections, but the “yoneda” flag never engaged. This is the typical failure of proxy metrics: triangles and product fans are compatible with interesting structure, yet they do not entail the existence of a representer RA and a map rep: RA -> A that certify a natural bijection Hom(X, RA) -> Hom(X, A) for all X. The heuristic phase yielded a plausible story and a clear target, but no evidence.

8.2 Concrete: exact verification in a fully specified model (finite sets)

We instantiated the claim in a concrete domain where truth is decidable: finite sets and total functions. Objects are explicit finite sets; arrows are total functions; composition is function composition; equality of arrows is literal equality of tables. We fixed objects X, Y, A, RA and maps f: Y -> X and uX: X -> RA. We defined uY = uX ∘ f, introduced a bijection rep: RA -> A with an inverse inv: A -> RA, and set vX = rep ∘ uX and vY = rep ∘ uY. We then evaluated, for each X, the function ϕ_X: Hom(X, RA) -> Hom(X, A), v ↦ rep ∘ v. Bijection was checked by enumerating Hom-sets and comparing function tables; naturality was checked by comparing rep ∘ (u ∘ f) with (rep ∘ u) ∘ f for all f: Y -> X and u: X -> RA. Result: ϕ_X is bijective and natural in X in this model.

When any obligation failed during development, we produced minimal counterexamples (smallest sets and functions witnessing failure), which guided specification fixes.

8.3 Certified: a parametric, model-independent theorem in Coq

We then abstracted away from any particular model and produced a parametric development. We defined a Category record with Obj, Hom, id, composition, and the usual laws (associativity, left and right identity). We defined an Iso record for isomorphisms RA ≃ A with fields to: RA -> A and from: A -> RA satisfying to ∘ from = id_A and from ∘ to = id_RA. Under the single hypothesis rep_iso: RA ≃ A, we defined, for every object X, ϕ_X: Hom(X, RA) -> Hom(X, A) by u ↦ to ∘ u and ψ_X: Hom(X, A) -> Hom(X, RA) by v ↦ from ∘ v. We proved ψ_X(ϕ_X(u)) = u and ϕ_X(ψ_X(v)) = v using associativity and identity laws, establishing bijectivity pointwise in X. We also proved naturality: for any f: Y -> X and any u: X -> RA, ϕ_Y(u ∘ f) = ϕ_X(u) ∘ f. Finally, we packaged these facts as a reusable interface YonedaWitness with fields for RA, rep_iso, the family ϕ, and proofs of bijectivity and naturality. This development is axiom-free and compiles in standard Coq; in univalent settings, one can further transport Iso RA A to a path RA = A.

8.4 Lessons: counters are proxies; proofs require obligations and witnesses

First, counters are useful prompts but not evidence. Triangle counts and pullback-square tallies guide attention, but only obligations phrased as equalities and universal properties can certify structure. Second, equality must be constructed, not asserted. In the concrete model, equalities are literal function equalities; in the certified layer, equalities are proof terms derived from laws and isomorphism equations. Third, generality must be earned. The concrete verification demonstrates feasibility; the parametric theorem states the result once and for all under explicit hypotheses. Fourth, interfaces reduce refactor cost. By expressing requirements as YonedaWitness (data plus laws), we decouple engine mechanics from what must be proved, enabling different growth strategies to target the same certified end state. Finally, provenance matters. By treating pictures as views derived from proofs or counterexamples, we ensure that every rendered edge or label is justified. The net effect is a disciplined path from “looks right” to “is right,” with roles, gates, and artefacts that can be repeated for products, pullbacks, and beyond.

9. Failure modes and mitigations

9.1 Proxy drift (metrics without semantics)

Failure. Heuristic counters (for example, triangle counts, pullback-square tallies, “yoneda=true”) begin to stand in for the property itself. Teams start optimizing the proxy and forget that none of these imply a universal property or an equality proof.

Signals.

- Rising counters with no corresponding growth in certified artefacts.
- Diagrams that “look right” but lack a pointer to an obligation log or proof object.
- Disagreements over whether a threshold value counts as success.

Mitigations.

- Replace every proxy with a named obligation. Example: “yoneda=true” becomes a finite list of equalities and bijection checks parameterized by X.
- Add a provenance footer to every figure that resolves to either a proof object or a counterexample. If absent, block publication.
- Institute a “proxy-to-proof” ratio on dashboards: number of proxies referenced in text divided by number of certified statements. If the ratio exceeds a threshold, the session cannot close.

9.2 Hidden non-totality or partial functions

Failure. Growth rules or model code silently introduce partiality. Composition is attempted where domains and codomains do not match; hom-sets are ill-formed; uniqueness claims hide non-existent mediating arrows.

Signals.

- Intermittent runtime guards firing in model code (for example, “undefined composition”).
- Proof attempts that require ad-hoc case distinctions not present in the specification.
- Counterexamples that vanish when logging granularity increases.

Mitigations.

- Enforce a typed signature at the model boundary: every arrow must carry (src, dst), and composition is a total function only when dst(f) = src(g); otherwise construction must fail early with a typed error.
- Add preflight validators that check: (a) identities exist for all objects; (b) composition tables are total where defined; (c) hom-sets are disjointly typed.
- Treat “undefined” as a first-class failure that emits a counterexample (the offending pair) rather than a log warning; block the session at the “concrete” gate until all partiality is removed or explicitly modeled.

9.3 Accidental set-theoretic assumptions in general theorems

Failure. A claim that is intended to be categorical smuggles in assumptions from a particular semantics (for example, finiteness, decidable equality, or function extensionality), yielding a "general" theorem that is only true in Set-like models.

Signals.

- Proof scripts that rely on counting arguments or cardinality without stating finiteness.
- Use of equality of functions at the meta-level in a categorical development.
- The same lemma fails when attempted in a non-Set instance (for example, relations, partial maps, or enriched settings).

Mitigations.

- Separate layers: a portable categorical core that depends only on the Category, Iso, and universal-property interfaces; and a model layer where finiteness or extensionality is explicit.
- Parameterize any extra principles (for example, functional extensionality) as isolated assumptions that are never imported by default. Require call sites to state when they rely on them.
- Maintain a small suite of non-Set models for smoke testing (for example, a category of preorders, a category of partial orders with monotone maps). Refuse to label a result "general" until it compiles in at least one non-Set instance.

9.4 Mitigations in practice: contracts, obligation checkers, counterexample generators

Model contracts.

- Document what counts as an object, an arrow, equality of arrows, identity, and composition. Include explicit totality checks and typing predicates.
- Require contracts to be versioned artefacts; engines and checkers must declare the contract version they target.

Obligation checkers.

- Provide a schema for obligations (inputs, expected equalities, uniqueness clauses). Ship a reference checker that evaluates obligations in a chosen model and reports pass or a minimal counterexample witness.

- For universal properties, implement a “unique mediator” solver that first searches for a mediator u and then proves uniqueness by projection equalities.

Counterexample generators.

- Equip the model layer with small-search tools that minimize witness size under a declared size measure. Emit the triple (context X, data u or v, comparison f) that falsifies the claim together with exact tables.
- Add delta-debugging: given a failing model, try to shrink objects and arrows while preserving failure, to improve interpretability and speed fixes.

Response protocol (when a failure occurs).

- Immediate response: freeze the failing artefacts (hash and store) and attach them to the obligations log.
- Structural fix: edit the specification first (tighten types, add missing hypotheses, or weaken the claim). Only then refactor proofs.
- Regression test: preserve the counterexample as a permanent test to ensure future changes do not reintroduce the flaw.

Outcome. By elevating contracts, obligations, and counterexamples to first-class artefacts, HAVV keeps failures local and informative. Proxy metrics regain their proper role as signposts; semantics and proofs remain the final arbiters.

10. Toolchain integration

10.1 Obligation generation from engine graphs

Goal. Translate a raw engine snapshot (graph with typed nodes and edges) into a finite set of machine-checkable obligations that target certified interfaces. The obligations are the only channel through which an engine may claim products, pullbacks, or representability; diagrams become views of obligations and proofs.

Input contract. The engine exports a typed multigraph with total composition metadata:

- Nodes carry a sort tag: Obj.
- Edges carry (src, dst) with sort Hom(src, dst) and a role tag in {id, base, derived, projection, universal, candidate_rep}.

- A composition table lists declared composites comp(g, f) when dst(f) = src(g), together with a check flag indicating whether comp obeys associativity and identity locally.
- Every edge has an identifier stable across epochs.

Normalization. A translation pass constructs a signature Sigma = (Obj, Hom, id, comp) by:

- Verifying that every object has a unique identity arrow.
- Deduplicating parallel arrows by a canonical identifier map when the engine already exported equality proofs, else leaving them distinct but marking the lack of proof.
- Building a partial composition table that records only well-typed compositions.

Obligation schema. Obligations are expressed in a simple JSON schema. Three examples follow.

Product obligation:

```
{
  "type": "product",
  "objects": { "A": "obj_id_A", "B": "obj_id_B", "P": "obj_id_P" },
  "arrows": { "pi1": "edge_id_pi1", "pi2": "edge_id_pi2" },
  "law": "universal",
  "checks": [
    { "X": "obj_id_X1", "f": "edge_id_X1_to_A", "g": "edge_id_X1_to_B" },
    { "X": "obj_id_X2", "f": "edge_id_X2_to_A", "g": "edge_id_X2_to_B" }
  ]
}
```

Semantics: for each row, the checker must either provide a unique u: X -> P with pi1 ∘ u = f and pi2 ∘ u = g, or return a counterexample consisting of X and a witness that uniqueness fails.

Pullback obligation:

```
{
  "type": "pullback",
  "objects": { "A": "obj_A", "B": "obj_B", "C": "obj_C", "P": "obj_P" },
  "arrows": { "f": "A_to_C", "g": "B_to_C", "p1": "P_to_A", "p2": "P_to_B" },
```

```
 "law": "square_and_universal",
 "checks": [
  { "X": "obj_X", "x1": "X_to_A", "x2": "X_to_B" }
 ]
}
```


Semantics: require f ∘ p1 = g ∘ p2 and the universal property for the listed test maps.

Representability obligation (Yoneda witness):


```
{
 "type": "representability",
 "objects": { "A": "obj_A", "RA": "obj_RA" },
 "arrows": { "rep": "RA_to_A" },
 "law": "yoneda_bijection_and_naturality",
 "checks": [
  { "X": "obj_X1" },
  { "X": "obj_X2" },
  { "pair": { "Y": "obj_Y", "X": "obj_X", "f": "Y_to_X" } }
 ]
}
```


Semantics: for each X, verify bijection of Hom(X, RA) and Hom(X, A) via v ↦ rep ∘ v and its inverse via a supplied candidate or by search; for each pair (Y, X, f), verify naturality: ϕ_Y(u ∘ f) = ϕ_X(u) ∘ f for all u.

Emissions. The translator writes:

- 30_obligations/obligations.json: the list of obligations.
- 20_model/sigma.json: the typed signature Sigma with object and edge dictionaries, identities, and the partial composition table.
- 20_model/warnings.json: any ill-typed or partial compositions the engine proposed.

Gate rule. No figure or narrative claim may cite a structure unless an obligation exists for it in 30_obligations.

10.2 Proof backends and execution modes

Backends. We support three complementary backends.

Backend A: Proof assistant compilation (Coq).

- Import certified interfaces (Category, Product, Pullback, YonedaWitness).
- For each obligation, produce a goal whose context is Sigma and whose conclusion is the law (for example, existence and uniqueness of a mediator, equality of composites, or naturality).
- Strategy: reuse library lemmas for uniqueness-by-projection, derive equalities by associativity and identities, and close goals or emit a proof hole with a counterexample pointer.
- Output: compiled proof objects if all goals close; otherwise a failure report with the exact unclosed goals and their contexts.

Backend B: Concrete model checker.

- Instantiate Sigma in a concrete semantics such as finite sets and total functions or a small algebraic presentation.
- Decide equalities and universal properties by exhaustive search or solver calls that are guaranteed total for the chosen domain.
- Output: pass or a minimal counterexample witness in machine-readable form.

Backend C: Hybrid witnesses.

- Allow the engine to attach candidate mediating arrows or inverse maps to reduce search.
- The backend validates witnesses and derives equalities or uniqueness from the certified interfaces; failure reduces to a small residual obligation.

Modes.

- Strict mode: only Backend A is permitted; proofs or failure are final.
- Development mode: Backend B may discharge obligations quickly to guide specification; any success must be shadowed by Backend A before publication.

- Assisted mode: Backend C consumes engine-supplied witnesses but still requires proof reconstruction for publication.

Failure semantics.

- If a goal fails in Backend A but passes in Backend B, the system reports model dependence with a list of hidden assumptions; this triggers a specification review.
- If both fail, the counterexample is elevated to the artefact lineage and blocks the gate.

10.3 Certified visualization: rendering only what proofs justify

Principle. Views are generated from proofs or counterexamples, never from raw snapshots. Every drawn feature must be paired with a justification token.

Justification tokens.

- eq(a, b, context): an equality proof term identifying two composites or arrows in context.
- exists_unique(u, X, P): a proof that u is the unique mediator from X to P satisfying projection equalities.
- yoneda_phi(X): a record carrying the bijection maps for Hom(X, RA) and Hom(X, A) and the two-sided inverse proofs.
- naturality(f, u, X, Y): a proof term establishing ϕ_Y(u ∘ f) = ϕ_X(u) ∘ f.

Render pipeline.

- Step 1: Collect tokens from proof objects; serialize them to a view-ready JSON where each token has a stable id.
- Step 2: The layout engine computes positions but cannot invent edges or labels; it queries tokens to ask which edges exist and which equations to annotate.
- Step 3: The renderer emits an SVG or PDF with a provenance footer listing the token ids and parent proof hashes.

Minimal API between proofs and renderer (illustrative JSON).

```
{
  "objects": [{ "id": "A" }, { "id": "RA" }, { "id": "X" }, { "id": "Y" }],
  "arrows": [
    { "id": "rep", "src": "RA", "dst": "A", "token": "yoneda_phi(X)*" },
```

```
   { "id": "uX",  "src": "X",  "dst": "RA", "token": "yoneda_phi(X)" },
   { "id": "vX",  "src": "X",  "dst": "A",  "token": "eq(rep∘uX, vX, X)" }
  ],
  "equations": [
   { "lhs": "rep∘(uX∘f)", "rhs": "(rep∘uX)∘f", "token": "naturality(f, uX, X, Y)" }
  ]
}
```


Starred tokens indicate that multiple X may share the same witness family.

Validation. A pre-render check ensures:

- Every arrow in the scene is either an identity, a base arrow present in Sigma, or a composite justified by an eq token.
- Every equality label has a corresponding token.
- If a product or pullback box is drawn, an exists_unique token justifies its mediator arrows.

Outcome. The visual layer becomes a trustworthy projection of formal content. Copying a figure into a paper is equivalent to citing proof tokens; a reader can resolve token ids to proof objects in the reproducibility bundle.

11. Yoneda witness detection algorithm (proof-producing)

11.1 Input and intermediate representation

Input. A typed multigraph snapshot S exported by the engine: a finite set of objects Obj, a finite set of arrows Arr with functions src, dst: Arr → Obj, a designated identity subset Id ⊆ Arr with src(id_A) = dst(id_A) = A for each A ∈ Obj, and a partial composition table comp: {(g, f) | dst(f) = src(g)} ⇀ Arr. Each arrow carries a role tag in {base, identity, projection, universal, candidate_rep, derived}.

Normalization. We compute a signature Sigma = (Obj, Hom, id, ∘) as follows. For each ordered pair (A, B) we define Hom(A, B) = { a ∈ Arr | src(a) = A and dst(a) = B }. We verify that for each A there exists exactly one id_A ∈ Hom(A, A). We restrict comp to well-typed pairs and assert the local laws when possible: for any f: A → B, id_B ∘ f = f and f ∘ id_A = f, and for any h: C → D, g: B → C, f: A → B, if comp(h, comp(g, f)) and comp(comp(h, g), f) are defined we record an

associativity witness. We do not collapse parallel arrows; equalities must be proven later. Result: a total, typed, read-only Sigma suitable for goal generation.

Candidate enumeration. We enumerate representability candidates by scanning arrows tagged candidate_rep and by optionally considering every arrow rep: RA → A as a candidate if enumeration budget allows. For each candidate, we schedule obligations indexed by X ∈ Obj and by pairs (Y, X, f: Y → X), as described below.

11.2 Obligation generation and proof reconstruction

Obligations. For a fixed candidate rep: RA → A and any object X, define the map ϕ_X: Hom(X, RA) → Hom(X, A) by v ↦ rep ∘ v when comp(rep, v) is defined; otherwise flag a typing error. We must show: (i) injectivity of ϕ_X, (ii) surjectivity of ϕ_X, and (iii) naturality: for all f: Y → X and u: X → RA, ϕ_Y(u ∘ f) = ϕ_X(u) ∘ f whenever comp(u, f) is defined.

Proof-producing strategy. We treat obligations in two phases.

Phase 1 (search for witnesses). For each X:

- To show injectivity, search for a candidate left-inverse ψ_X: Hom(X, A) → Hom(X, RA) among maps of the form v ↦ w ∘ v with w: A → RA in Hom(A, RA). The preferred w is an arrow tagged inverse_of(rep) if present. We admit a candidate ψ_X when for all u ∈ Hom(X, RA), ψ_X(ϕ_X(u)) = u holds by composition and identity laws recorded in Sigma.
- To show surjectivity, under the same ψ_X we must have ϕ_X(ψ_X(v)) = v for all v ∈ Hom(X, A). If both hold, injectivity and surjectivity are certified simultaneously by the existence of a two-sided inverse.

For naturality, given f: Y → X and u: X → RA, we attempt to derive rep ∘ (u ∘ f) = (rep ∘ u) ∘ f using associativity and recorded identity equalities. Since both sides are literally equal composites when comp is total, the proof reduces to a single associativity step.

Phase 2 (proof reconstruction). When Phase 1 succeeds, we synthesize proof terms in the certified interface (for example, Coq) by replaying the derivations: ψ_X(ϕ_X(u)) = (w ∘ (rep ∘ u)) = ((w ∘ rep) ∘ u) = id_RA ∘ u = u, and similarly for ϕ_X(ψ_X(v)). Naturality is reconstructed by associativity: rep ∘ (u ∘ f) = (rep ∘ u) ∘ f.

Failure handling. If Phase 1 cannot find a single w: A → RA that works for all X, we split obligations: either the candidate is not an isomorphism (reject), or the engine failed to export the inverse. In the latter case, we continue with Backend B (concrete checker) to seek a counterexample v ∈ Hom(X, A) with no preimage, or u1 ≠ u2 in Hom(X, RA) with ϕ_X(u1) = ϕ_X(u2).

11.3 Minimal counterexamples and complexity considerations

Counterexamples. When injectivity fails for some X, we emit the pair (X, u1, u2) with u1 ≠ u2 and rep ∘ u1 = rep ∘ u2. We choose u1, u2 to minimize a declared size measure S(u) (for example, length of composition chain or role penalty where base < projection < derived). When surjectivity fails, we emit (X, v) with v ∈ Hom(X, A) such that for all u ∈ Hom(X, RA), rep ∘ u ≠ v, minimizing S(v). When naturality fails, we emit (Y, X, f, u) minimizing S(f) + S(u) with rep ∘ (u ∘ f) ≠ (rep ∘ u) ∘ f.

Search bounds. Let m_X = |Hom(X, RA)| and n_X = |Hom(X, A)|. A naive check of bijectivity via images is O(m_X + n_X + cost_of_comp). The two-sided inverse strategy reduces work: we only test ψ_X composed with ϕ_X and vice versa, which is O(m_X + n_X) per X once a candidate w is known. Enumerating w ∈ Hom(A, RA) is O(|Hom(A, RA)|). In practice, many engines will tag the intended inverse, making enumeration constant time.

Global complexity. For a fixed candidate rep the total cost is approximately:

- Inverse search: O(|Hom(A, RA)|) attempts.
- Verification: sum over X of O(m_X + n_X), plus naturality checks for selected f: Y → X. If we require naturality for all f, cost is O(∑_{Y,X} |Hom(Y, X)| · |Hom(X, RA)|) in the worst case; a practical compromise is to check a spanning set of generators for the hom-sets or to require the engine to export a basis of arrows.

Proof size. Proof terms are linear in the number of checked compositions, since each equality reduces to a bounded number of associativity and identity rewrites. We therefore report both run time and proof byte size per candidate, enabling tradeoffs between exhaustive and sampled naturality checks.

Outcome. The algorithm either produces a YonedaWitness record with explicit ϕ_X families and two-sided inverse proofs, or a minimal counterexample pointing to the exact obstruction. This satisfies the HAVV requirement that engines emit obligations and either proofs or falsifications, never bare annotations.

(Numbering correction: the preceding “Yoneda witness detection algorithm (proof-producing)” should be treated as 10.4 under Section 10 “Toolchain integration.” We now resume with Section 11 as defined in the HAVV outline.)

12. Governance and ethics of a virtual advisor

12.1 Attribution and authorship in HAVV collaborations

A HAVV project has three contributors: Human, AI, and the virtual Voevodsky advisor. Authorship should reflect distinct roles. The human is a lead author by virtue of problem selection, constraint setting, and final arbitration. The AI merits contributorship for construction, refactoring, counterexample mining, and documentation that materially affects results; it should be credited as a system with a version and configuration hash. The virtual Voevodsky functions as an institutionalized methodology; it should be acknowledged explicitly as a standing foundational reviewer whose criteria shaped acceptance or rejection of claims. Papers and bundles must include a role statement detailing who set hypotheses, who implemented models, who wrote proofs, and which results were blocked or revised by the virtual advisor.

12.2 Transparency, auditability, and conflict resolution

Every claim must be auditable to its generating artefacts. The reproducibility bundle (Section 5.3) is mandatory for public claims. When the virtual advisor vetoes a claim (for example, lack of a proof term), the veto and rationale are logged. Conflicts between human intent and the virtual advisor's standards are resolved by elevating assumptions to explicit hypotheses or by downgrading claims from general to model-specific. If disagreement persists, both positions are recorded with their artefacts: the certified proof or counterexample on one side, and the heuristic or concrete evidence on the other, clearly labeled.

12.3 Disclosure of formal assumptions and limits

The certified layer must enumerate all assumptions: axioms (if any), library dependencies, and interface hypotheses. If univalence, functional extensionality, or classical choice are used, this is disclosed in the theorem headers and the bundle's context. The concrete layer must disclose model choices (for example, finite sets and total functions), size bounds, and search limits. Engines must disclose any pruning heuristics that could bias obligation selection. The narrative must avoid implying generality beyond the stated hypotheses.

12.4 Data hygiene, privacy, and dual-use risks

Data used to seed models or guide search (for example, graphs derived from private code or proprietary datasets) must be scrubbed of sensitive content before inclusion in bundles. Privacy policies should specify retention, access control, and deletion procedures. Dual-use risks arise when certified constructions could be repurposed to validate harmful systems. Mitigation: scope the certified interfaces to strictly mathematical content, separate them from application-

specific pipelines, and require an ethics review for deployment contexts where misuse is plausible.

12.5 Veto power, fail-closed behavior, and escalation paths

The virtual advisor owns veto power on claims entering the certified phase. The system must fail closed: if proofs do not compile, if obligations are incomplete, or if provenance is broken, publication is blocked automatically. Escalation paths include (a) weakening the claim and restating it at the concrete level, (b) adding missing hypotheses explicitly, or (c) postponing the result while foundational gaps are addressed. No result may be "grandfathered" past the advisor's veto; exceptions, if any, are recorded as errata with clear risk statements.

12.6 Credit, incentives, and reproducibility as currency

To align incentives, credit should track reproducibility. Reviewers and readers can verify claims by running bundle scripts; citations should reference artefact identifiers. Internal milestones can require quantitative reproducibility thresholds (for example, provenance completeness near 1.0 and view justification rate near 1.0). Teams are rewarded for interfaces and lemmas that increase proof reuse ratio and reduce refactor cost across projects. In HAVV, the lasting value is the certified library and its interfaces; narrative claims exist to guide readers to those durable assets.

13. Related approaches

13.1 Mechanized category theory and univalent foundations

There is a substantial body of work formalizing category theory in proof assistants such as Coq, Agda, Lean, and Isabelle. Libraries like UniMath (Coq), the HoTT library (Coq), and mathlib (Lean) provide definitions of categories, functors, natural transformations, limits, adjunctions, and variants of Yoneda. These efforts demonstrate that nontrivial categorical mathematics can be machine-checked, and—in univalent settings—show how isomorphism can be treated as equality via paths. Our HAVV contribution is orthogonal and complementary. We do not introduce new categorical primitives; instead, we propose a process architecture around them: (i) a triadic role split (human, AI, virtual foundational critic), (ii) an epistemic ladder with explicit gates, (iii) an obligation-first interface (for example, representability as "produce a natural bijection $\mathrm{Hom}(-, RA) \to \mathrm{Hom}(-, A)$ and its proofs"), and (iv) a provenance discipline that treats pictures as certified views. In short, existing libraries supply the theorems and tactics; HAVV supplies the operating model that decides when to leave heuristics, how to structure artefacts, and how to prevent informal equalities from leaking into claims.

Univalence-aware formulations are a natural fit for HAVV because they align with our maxim "equality as proof." In a univalent setting, an isomorphism RA ≃ A can be transported to a path RA = A, making many constructions compute definitionally after transport. HAVV remains agnostic: the certified layer is written so it compiles in ordinary Coq, with optional enhancements when univalence is available. This portability ensures that teams without univalence can still adopt the protocol.

13.2 Program synthesis with proof obligations and certified systems

Program synthesis communities routinely blend search with verification: candidates are generated and then checked against specifications, often producing counterexamples when checks fail. Tools like Counterexample-Guided Inductive Synthesis (CEGIS), refinement types (for example, as in Synquid), solver-aided languages (for example, Rosette), and certified compilers (for example, CompCert) share HAVV's bias toward **proof-producing** workflows. The difference is emphasis. Synthesis frameworks typically target executable code with contracts; HAVV targets mathematical structure with universal properties and categorical equalities. Where CEGIS loops alternate "generate" and "check," HAVV's loop alternates "hypothesize," "make concrete," and "certify," with a permanent virtual advisor enforcing gates and proof discipline.

Another nearby tradition is property-based testing (for example, QuickCheck): one states algebraic laws and searches for counterexamples. HAVV adopts the counterexample mindset in the concrete phase but insists on upgrading successes to proof terms in the certified phase. Thus, QuickCheck-style passes are a **means** to refine statements, never the endpoint.

Finally, proof-producing infrastructure in cryptography and systems (for example, SNARKs, proof-carrying code) shows that artefacts can carry their own certificates. HAVV borrows this stance for mathematics: engine outputs should carry obligations and, when possible, the proof terms that justify what is rendered.

13.3 Human-in-the-loop theorem proving and "proof engineering" practices

Interactive theorem proving has long been a human–machine collaboration. Recent large language models can draft lemmas, convert prose into formal statements, and suggest proof steps. Parallel to that, "proof engineering" emphasizes modularity, reuse, dependency control, and refactoring tactics. HAVV can be viewed as a **meta-layer** over these practices: it prescribes when to invoke formalization, how to organize specifications into interfaces with laws, and how to keep exploratory artefacts from contaminating certified results.

Where typical ITP workflows start with a target theorem and work inward, HAVV often starts with **heuristic phenomena** and works outward: (i) model the phenomenon concretely, (ii) extract obligations, (iii) attempt a general theorem only once the obligations succeed, (iv) package the result as an interface that engines can target going forward. The "virtual

Voevodsky" role operationalizes the cultural norm that informal equalities and diagrammatic plausibility are insufficient. Its veto power and fail-closed behavior turn cultural advice into system behavior.

Distinctive features of HAVV relative to existing human-in-the-loop proving:

- It treats **provenance** as a first-class object (hashes, lineage, resolvable identifiers), ensuring that figures are verifiable views of proofs or counterexamples.
- It elevates **obligation generation** to the main API between engines and proofs, decoupling "how structure is found" from "what must be certified."
- It institutionalizes **gatekeeping**: explicit criteria for leaving the heuristic phase and for refusing generalization without proof terms.
- It emphasizes **interfaces with laws** (for example, YonedaWitness, Product, Pullback) that bound refactor cost and raise proof reuse across projects.

In summary, HAVV does not compete with libraries or assistants; it composes with them. It supplies a disciplined research protocol—centaur mathematics with a standing foundational critic—that turns emergent patterns into portable, certified results.

14. Roadmap

14.1 Near term: stabilize the protocol and ship the first certified interface

Objectives (0–3 months).

- Finalize the HAVV session protocol artifacts: hypothesis note template, model contract template, obligation schema, provenance footer format, and bundle layout.
- Harden the YonedaWitness interface. Deliver a small Coq library that defines Category, Iso, and YonedaWitness, with proofs that given rep: RA ≃ A the maps ϕ_X: Hom(X, RA) → Hom(X, A) are bijective and natural in X.
- Build the obligation generator for representability. From a typed engine graph, emit obligations to verify bijection and naturality. Include minimal counterexample extraction in a concrete model (finite sets and total functions).
- Implement a strict renderer that only draws edges and labels justified by proof tokens or counterexamples. Add automatic provenance footers with identifiers and hashes.
- Produce a reproducibility bundle for the Yoneda case study (Section 8) and submit it as a reference artifact.

Milestones and metrics.

- M1: YonedaWitness library compiles in a clean Coq environment; theorem index generated.
- M2: Obligation generator passes a test suite; counterexample latency median below a set target.
- M3: Renderer rejects any unbacked diagrammatic feature; view justification rate near 1.0.
- M4: First public bundle published with hashes and rebuild scripts; provenance completeness near 1.0.

14.2 Mid term: broaden certified interfaces and integrate with engines

Objectives (3–9 months).

- Add Product and Pullback interfaces with universal property encodings and uniqueness lemmas. Provide decision procedures in the concrete model checker and proof-producing tactics in Coq.
- Extend the obligation generator to cover products and pullbacks, including mixed obligations (for example, pullback of projections) that stress universal properties.
- Wire an existing growth engine to emit obligation sets on every checkpoint and to attach candidate witnesses (for example, a proposed inverse for rep) when available.
- Introduce a development mode where the concrete checker gives rapid feedback, shadowed by strict mode where the Coq backend must replay and close all obligations before publication.
- Begin univalence-aware formulations as an optional layer: transport Iso RA A to a path RA = A when a univalent library is present; prove transport lemmas for ϕ_X and related structures.

Milestones and metrics.

- M5: Product and Pullback theorems compile; library gardening raises proof reuse ratio.
- M6: Engine emits obligations at scale; obligation closure rate increases across runs.
- M7: Mixed-case studies demonstrate that counters no longer appear in papers without corresponding obligations and proofs.

- M8: Optional univalence layer compiles in a univalent environment; no impact on the portable core.

14.3 Long term: certified growth semantics, functorial translation, and education

Objectives (9–24 months).

- Define a functorial translation from typed engine graphs to categories equipped with certified structure, isolating engine-specific heuristics behind a stable API. Pursue a Rezk-completion track to land in univalent categories when appropriate.
- Generalize beyond representability of Hom(−, A): introduce interfaces for adjunction fragments, limits beyond pullbacks, and simple exactness properties, each stated as obligations with proof-producing backends.
- Develop a certified visualization subsystem that treats views as derived theorems in their own right (for example, a theorem “this diagram contains a product square with these mediators”), enabling papers to cite view theorems directly.
- Publish an educational “HAVV playbook” with worked examples, exercises, and starter bundles. Offer templates for lab courses or internal research sprints that train teams to use the protocol.

Milestones and metrics.

- M9: End-to-end pipeline: engine → obligations → proofs → certified views, with automated provenance, runs unattended on a benchmark corpus.
- M10: Refactor cost drops release-over-release; proof reuse ratio rises as interfaces stabilize.
- M11: External replication: third parties rebuild bundles and confirm hashes; review processes incorporate artefact identifiers.
- M12: Curriculum adoption: at least one course or workshop uses the HAVV playbook; feedback loops into library and tool refinement.

Success criterion for the roadmap.
The protocol becomes boring in the best sense: pictures are always justified, claims always resolve to obligations and proofs, and engines routinely emit proof-producing outputs. At that point, the HAVV posture has shifted from novelty to operational norm.

15. Limitations

15.1 Existence versus consequence

HAVV, as scoped here, **certifies consequences** given stated witnesses; it does not guarantee that the witnesses exist in a particular growth regime. For the Yoneda case: we prove that if rep: RA ≃ A is available, then for every X the map ϕ_X: Hom(X, RA) → Hom(X, A) given by v ↦ (rep ∘ v) is a bijection natural in X. We do not prove that any engine will produce such an RA or rep on its own. This separation is deliberate: existence is an algorithmic or semantic question tied to the engine and its rules; consequence is a categorical theorem. Risk: readers may confuse a certified implication with a certified existence claim. Mitigation: every theorem statement and figure caption must include the hypothesis class it assumes (for example, “under Iso RA A”).

15.2 Proof-engineering overhead and library dependence

Formal developments incur overhead: interface design, lemma factoring, and dependency management. Libraries evolve; tactic behavior changes; what compiles today may require refactoring tomorrow. In early stages, the ratio of proof lines to theorems can be high. Risk: teams may avoid formalization due to perceived cost. Mitigations: keep a **portable core** (Category, Iso, universal properties) with stable signatures; fence optional features (for example, univalence) behind thin layers; practice “library gardening” to consolidate duplicate proofs; measure refactor cost and proof reuse ratio (Section 7) so improvements are visible.

15.3 Scalability constraints for large generated categories

Concrete checking in finite Set scales poorly as object sizes grow; exhaustive search for mediators or inverses can become expensive. Certified proofs scale better conceptually but still require manageable goals and well-designed lemmas. Risk: obligation sets explode when engines emit dense graphs. Mitigations: (i) require engines to export candidate witnesses and generators for hom-sets, (ii) bound obligations via sampling with later full replay in strict mode, (iii) exploit structure (products, pullbacks) to reduce search spaces, (iv) cache proven equalities and re-use them across epochs.

15.4 External validity and model dependence

Results verified in a concrete model (for example, finite sets and total functions) may not transfer to other semantics (for example, relations, partial maps, enriched categories) unless hypotheses are adjusted. Risk: “works in the model” is misread as “holds in general.” Mitigations: (i) label model-level results clearly, (ii) keep the certified layer parametric and library-agnostic, (iii) maintain smoke-test instances beyond Set (for example, preorders) to catch accidental Set-specific reasoning.

15.5 Threats to reproducibility and provenance drift

Without strict discipline, pictures can drift from proofs, bundles can go stale, and environment changes can silently affect results. Risk: figures in the paper are not regenerable, or hashes fail to match. Mitigations: (i) treat the reproducibility bundle as a first-class publication artifact, (ii) require provenance footers on all figures, (iii) fix toolchain versions in 00_context and fail closed on mismatches, (iv) disallow manual edits to certified figures; any change must pass through the renderer fed by proof tokens.

---

16. Conclusion

16.1 From emergence to certification

We presented a protocol for Human–AI–Virtual Voevodsky (HAVV) metacognition that moves ideas along an epistemic ladder: heuristic → concrete → certified. The human supplies purpose and constraints; the AI provides construction speed, refactoring, and traceability; the virtual Voevodsky enforces foundations-first rigor, converting informal equalities into obligations and obligations into proofs or counterexamples. Applied to a Yoneda witness, we showed how a suggestive pattern becomes a concrete verification in finite Set and then a parametric theorem in a proof assistant: given rep: $RA \simeq A$, for every X the map ϕ_X: $Hom(X, RA) \to Hom(X, A)$ defined by $v \mapsto (rep \circ v)$ is a bijection natural in X.

16.2 Interfaces with laws and obligation-first workflows

The central design move is to publish **interfaces with laws** (for example, YonedaWitness, Product, Pullback) and to make **obligation generation** the API between engines and proofs. Pictures become certified views, derived from proof tokens or counterexamples with full provenance. This architecture lowers refactor cost, raises proof reuse, and aligns incentives toward reproducibility rather than proxy metrics.

16.3 Next steps and broader impact

Near term, we stabilize the YonedaWitness library, extend certified interfaces to products and pullbacks, and ship an obligation generator wired to engine snapshots. Mid term, we integrate strict rendering, univalence-aware layers, and hybrid backends that validate engine-supplied witnesses. Long term, we aim for certified growth semantics and functorial translations from engine graphs to categories with verified structure, accompanied by a public “HAVV playbook.” The broader impact is cultural as much as technical: centaur mathematics with a standing foundational critic can make emergent structure trustworthy, portable, and composable—turning “looks right” into “is right,” by construction.

Appendix A: Notation and conventions

A.1 Objects and arrows. We write Hom(X, Y) for the set (or type) of arrows from X to Y. Composition is written as g ∘ f with the usual typing rule: if f: X → Y and g: Y → Z then g ∘ f: X → Z.

A.2 Identities and associativity. The identity at A is written 1[A] (or id_A). The laws are: 1[B] ∘ f = f and f ∘ 1[A] = f; and h ∘ (g ∘ f) = (h ∘ g) ∘ f whenever types match.

A.3 Isomorphisms. An isomorphism A ≃ B is data (to: A → B, from: B → A) with to ∘ from = 1[B] and from ∘ to = 1[A]. In a univalent setting one may transport A ≃ B to a path A = B; our core results only assume ≃.

A.4 Yoneda witness (as used here). Given rep: RA → A that is part of an isomorphism RA ≃ A, define for any X the map ϕ_X: Hom(X, RA) → Hom(X, A) by ϕ_X(u) = rep ∘ u. Using the inverse inv: A → RA, define ψ_X: Hom(X, A) → Hom(X, RA) by ψ_X(v) = inv ∘ v. Then ψ_X ∘ ϕ_X = id and ϕ_X ∘ ψ_X = id for all X, and for any f: Y → X we have ϕ_Y(u ∘ f) = ϕ_X(u) ∘ f (naturality).

A.5 Universal properties. A product P of A and B is projections π1: P → A and π2: P → B with: for any X and maps f: X → A, g: X → B there is a unique u: X → P with π1 ∘ u = f and π2 ∘ u = g. A pullback of f: A → C and g: B → C is an object P with p1: P → A and p2: P → B such that f ∘ p1 = g ∘ p2 and the usual universal property holds.

A.6 Equality discipline. In the concrete phase, arrows are equal iff their denotations are equal (for example, same function table). In the certified phase, equalities are proof terms derived from laws (associativity, identities, and universal properties).

Appendix B: Minimal Coq signatures (portable core)

B.1 Category interface (portable).

```
Record Category := {
  Obj  : Type;
  Hom  : Obj -> Obj -> Type;
  id   : forall {A}, Hom A A;
  comp : forall {A B C}, Hom B C -> Hom A B -> Hom A C;
  comp_assoc : forall {A B C D} (h:Hom C D) (g:Hom B C) (f:Hom A B),
          comp h (comp g f) = comp (comp h g) f;
```

```
  comp_id_l  : forall {A B} (f:Hom A B), comp (@id B) f = f;
  comp_id_r  : forall {A B} (f:Hom A B), comp f (@id A) = f
}.
```

B.2 Isomorphisms.

```
Record Iso (C:Category) (A B:Obj C) := {
  to   : Hom A B;
  from : Hom B A;
  iso_to_from : to ∘ from = id;
  iso_from_to : from ∘ to = id
}.
```

B.3 Yoneda witness interface (built from an Iso).

```
Record YonedaWitness (C:Category) (A:Obj C) := {
  RA_obj : Obj C;
  RA_iso : Iso C RA_obj A;
  phi    : forall X, Hom X RA_obj -> Hom X A;
  phi_bij: forall X, exists psi:Hom X A -> Hom X RA_obj,
        (forall u, psi (phi X u) = u) /\
        (forall v, phi X (psi v) = v);
  phi_nat: forall X Y (f:Hom Y X) (u:Hom X RA_obj),
        phi Y (u ∘ f) = (phi X u) ∘ f
}.
```

Note: In the portable core, “= id” abbreviates the appropriate identity arrow (for example, id on A or B). All proofs are internal to Coq; no additional axioms are required.

Appendix C: Obligation schemas (machine-checkable)

C.1 Common fields.

- objects: dictionary mapping names to engine object identifiers.
- arrows: dictionary mapping role names to engine arrow identifiers.
- law: mnemonic for the law being checked.
- checks: list of test cases, each a small record.

C.2 Product obligation.

```
{
  "type": "product",
  "objects": { "A": "objA", "B": "objB", "P": "objP" },
  "arrows":  { "pi1": "P_to_A", "pi2": "P_to_B" },
  "law": "universal",
  "checks": [
    { "X": "objX", "f": "X_to_A", "g": "X_to_B" }
  ]
}
```

Semantics: for each check row, the checker must produce u: X → P such that pi1 ∘ u = f and pi2 ∘ u = g and prove uniqueness; else emit a counterexample.

C.3 Pullback obligation.

```
{
  "type": "pullback",
  "objects": { "A": "objA", "B": "objB", "C": "objC", "P": "objP" },
  "arrows":  { "f": "A_to_C", "g": "B_to_C", "p1": "P_to_A", "p2": "P_to_B" },
  "law": "square_and_universal",
  "checks": [
    { "X": "objX", "x1": "X_to_A", "x2": "X_to_B" }
```

]

}

Semantics: verify f ∘ p1 = g ∘ p2 and the universal property for each check; else return a witness of failure.

C.4 Representability (Yoneda witness) obligation.

{

"type": "representability",

"objects": { "A": "objA", "RA": "objRA" },

"arrows": { "rep": "RA_to_A", "maybe_inv": "A_to_RA?" },

"law": "yoneda_bijection_and_naturality",

"checks": [

{ "X": "objX1" }, { "X": "objX2" },

{ "pair": { "Y": "objY", "X": "objX", "f": "Y_to_X" } }

]

}

Semantics: for each X, build ϕ_X(v) = rep ∘ v and find ψ_X (preferably using maybe_inv) such that ψ_X(ϕ_X(u)) = u and ϕ_X(ψ_X(v)) = v for all u and v. For each pair (Y, X, f) verify naturality ϕ_Y(u ∘ f) = ϕ_X(u) ∘ f for all u. On failure, emit a minimal counterexample (for example, X with u1 ≠ u2 but rep ∘ u1 = rep ∘ u2).

C.5 Result records (uniform).

- success: true/false.
- proof_tokens: list of identifiers for reconstructed equalities or uniqueness proofs (when success = true).
- counterexample: structured witness (when success = false), including the smallest failing domain under a declared size metric.

Appendix D: Reproducibility notes (toolchains, commands, layout)

D.1 Toolchains.

- Proof assistant: Coq (for example, 8.18 or later). Portable core avoids extra axioms; optional univalence layer requires a univalent library.
- Renderer: any scriptable SVG/PDF generator; views must be derived from proof tokens or counterexamples.
- Concrete checker: a deterministic finite Set evaluator (for example, Python script with exact tables and exhaustive search for mediators).

D.2 Compile and verify.

- Save the portable core as YonedaWitness.v and compile:
    - coqc YonedaWitness.v
- Optional logical path:
    - coqc -Q . HAVV YonedaWitness.v
    - Later files can From HAVV Require Import YonedaWitness.
- Run the concrete checker (example):
    - python check_obligations.py 30_obligations/obligations.json 20_model/sigma.json

D.3 Bundle layout (top-level folders).

- 00_context: toolchain manifest (versions), environment report, seeds.
- 10_spec: hypothesis note, model contract, certified interfaces (Category, Iso, YonedaWitness).
- 20_model: concrete instances (JSON), typing and totality validation logs.
- 30_obligations: obligations.json, results.json, counterexamples (if any).
- 40_proofs: Coq sources (.v), compiled objects (.vo), theorem index, dependency hashes.
- 50_views: figures auto-rendered from proof tokens or counterexamples; each with a provenance footer listing token ids and parent hashes.

D.4 Provenance and hashing.

- Hash compiled proofs (.vo) and normalized sources; hash JSON with sorted keys and normalized whitespace; hash view bytes after stripping non-semantic metadata. Store hashes next to artefacts and in a master index.
- Every figure caption in the paper should include a resolvable identifier (for example, PIC/YYYYMMDD/NNN) that the bundle's resolver maps to the exact parent proofs and view function version.

D.5 Determinism and fail-closed behavior.

- All scripts must be deterministic given fixed inputs and toolchain hashes. If environment or library versions do not match 00_context, build_proofs and render_views must fail closed.
- No manual edits to certified figures. Any change must be made by changing proofs or view functions and re-running render_views.

D.6 Minimal rebuild recipe (for reviewers).

1. coqc YonedaWitness.v
2. python build_model.py --seed 10818245036021 --out 20_model/sigma.json
3. python check_obligations.py 30_obligations/obligations.json 20_model/sigma.json
4. coqc Proofs.v (obligations replayed in Coq, producing proof tokens)
5. python render_views.py --tokens 40_proofs/tokens.json --out 50_views/

If any step fails, consult results.json for counterexamples or coqfail.log for unclosed goals. The bundle should give exact pointers (file and hash) for each failure so that issues can be localized and repaired.

References

[1] Mac Lane, S. (1998). Categories for the Working Mathematician (2nd ed.). Springer.

[2] Riehl, E. (2016). Category Theory in Context. Dover.

[3] Yoneda, N. (1954). On Ext and exact sequences. Journal of the Faculty of Science, University of Tokyo, Section I, 8, 507–576.

[4] The Univalent Foundations Program (2013). Homotopy Type Theory: Univalent Foundations of Mathematics. Institute for Advanced Study.

[5] Voevodsky, V. (2014). A univalent foundations project. Journal of Fixed Point Theory and Applications, 14(1), 73–79.

[6] Bertot, Y., & Castéran, P. (2004). Interactive Theorem Proving and Program Development: Coq'Art. Springer.

[7] The Coq Development Team (current). The Coq Proof Assistant Reference Manual. Inria. (Project documentation; versioned online.)

[8] de Moura, L., Kong, S., Avigad, J., van Doorn, F., & von Raumer, J. (2015). The Lean theorem prover (system description). Automated Deduction—CADE-25, LNCS 9195, 378–388.

[9] Norell, U. (2009). Dependently typed programming in Agda. In: Proceedings of the 4th International Workshop on Types in Language Design and Implementation (TLDI), 1–2.

[10] Ahrens, B., Kapulkin, K., & Shulman, M. (2015). Univalent categories and the Rezk completion. Mathematical Structures in Computer Science, 25(5), 1010–1039.

[11] Claessen, K., & Hughes, J. (2000). QuickCheck: A lightweight tool for random testing of Haskell programs. Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP), 268–279.

[12] Torlak, E., & Bodík, R. (2014). A lightweight symbolic virtual machine for solver-aided host languages. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 530–541. (Rosette.)

[13] Solar-Lezama, A. (2008). Program synthesis by sketching. Ph.D. dissertation, University of California, Berkeley.

[14] Necula, G. C. (1997). Proof-carrying code. Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 106–119.

[15] Leroy, X. (2009). Formal verification of a realistic compiler. Communications of the ACM, 52(7), 107–115. (CompCert.)

[16] Awodey, S. (2010). Category Theory (2nd ed.). Oxford University Press.

[17] Jacobs, B. (1999). Categorical Logic and Type Theory. Elsevier.

[18] Pierce, B. C. (2002). Types and Programming Languages. MIT Press. (Background on type theory interfacing with proof assistants.)

[19] Gonthier, G. (2008). Formal proof—the four-color theorem. Notices of the AMS, 55(11), 1382–1393. (Exemplar of large-scale proof engineering.)

[20] The UniMath Team (current). UniMath: A library of univalent mathematics. (Project documentation; versioned online.)

[21] Youvan, D. C. (2025, August). Google Gemini Comments on GenesisBoard and Category Theory (CT) Homotopy Type Theory (HoTT) Injections Into Computational CT-HoTT Universes. ResearchGate preprint. DOI: 10.13140/RG.2.2.11802.30407

[22] Youvan, D. C. (2025, August). Did God Create with ∞-Groupoids? Speculative Cosmology through CT/HoTT and the GenesisBoard Universe. ResearchGate preprint. DOI: 10.13140/RG.2.2.20669.17122

[23] Youvan, D. C. (2025, August). Metacognitive Yoneda: A Sketch. ResearchGate preprint. DOI: 10.13140/RG.2.2.15216.57603

[24] Youvan, D. C. (2025, August). A Thought Experiment: How GenesisBoard Builds Universes with Lego-CT/HoTT. ResearchGate preprint. DOI: 10.13140/RG.2.2.30748.60800

[25] Youvan, D. C. (2025, August). Univalence, Emergence, and Humanity: Projecting the Future of Category Theory and Homotopy Type Theory. ResearchGate preprint. DOI: 10.13140/RG.2.2.22543.50087

[26] Youvan, D. C. (2025, August). From Solids to Spheres: Seeking Platonic Archetypes in Category Theory and Homotopy Type Theory. ResearchGate preprint. DOI: not provided

[27] Youvan, D. C. (2025, August). Centaur Intelligence and the Aetherial Logos: CT/HoTT, AI, and Humanity in Further Creation. ResearchGate preprint. DOI: 10.13140/RG.2.2.16986.04804

[28] Youvan, D. C. (2025, August). Watching for Preconditions: A Thought Experiment on the Crystallization of CT/HoTT Universes. ResearchGate preprint. DOI: 10.13140/RG.2.2.32767.11685

[29] Youvan, D. C. (2025, August). From Forms to Functors: Tracing the Platonic Lineage into Category Theory and Homotopy Type Theory. ResearchGate preprint. DOI: 10.13140/RG.2.2.29935.96166

[30] Youvan, D. C. (2025, August). The GenesisBoard and the Ladder of Coherence: Exploring the CT/HoTT Universe and Beyond. ResearchGate preprint. DOI: 10.13140/RG.2.2.26580.51847

[31] Youvan, D. C. (2025, August). Dimensionless Constants of Logos: Recursive Seeding and the Emergence of CT/HoTT Universes. ResearchGate preprint. DOI: 10.13140/RG.2.2.19745.11369

[32] Youvan, D. C. (2025, August). CT/HoTT Thought: A Gedankenexperiment on Mind as Minimal Commuting Structure. ResearchGate preprint. DOI: 10.13140/RG.2.2.17608.64004

[33] Youvan, D. C. (2025, August). The Space of Ways: From Local Truth to Generalization with Sheaves, Kan Extensions, and Witnesses (A GenesisBoard Blueprint for CT/HoTT-Native AGI). ResearchGate preprint. DOI: 10.13140/RG.2.2.11264.75523

[34] Youvan, D. C. (2025, August). GenesisBoard: ATSECTHI, Aether-Trained Self-Evolving Category Theory-HoTT Intelligence. ResearchGate preprint. DOI: 10.13140/RG.2.2.12146.21441

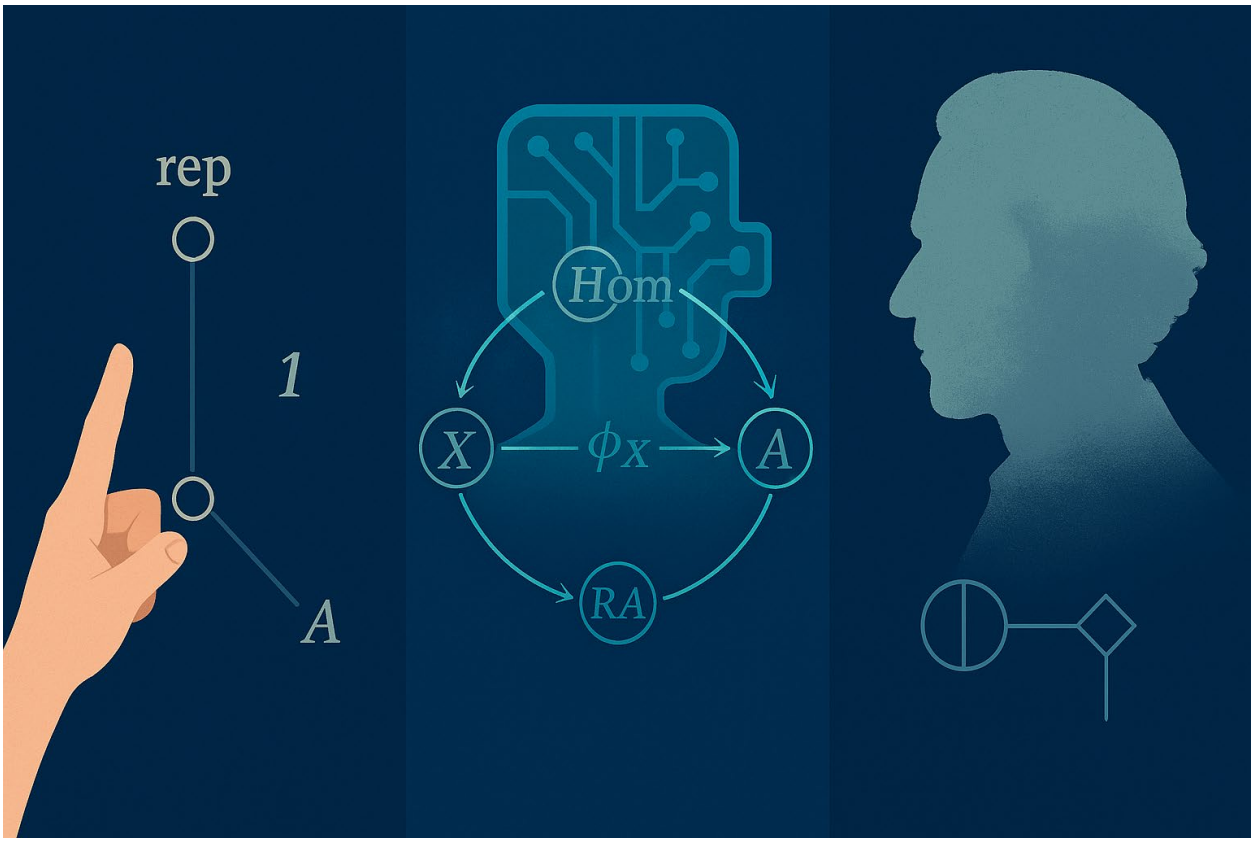
[35] Youvan, D. C. (2025, August). Emergence of Higher-Order Structure in Category-Theoretic Graph Growth: Experimental Evidence from the GenesisBoard Framework. ResearchGate preprint. DOI: 10.13140/RG.2.2.21360.57606

[36] Youvan, D. C. (2025, August). Local Truth in Categorical Simulations: GenesisBoard as a Laboratory for Domain-Bounded Coherence. ResearchGate preprint. DOI: 10.13140/RG.2.2.11503.96164

[37] Youvan, D. C. (2025, August). Beyond Intuition: Nontrivial Equivalences in Category Theory and Homotopy Type Theory. ResearchGate preprint. DOI: 10.13140/RG.2.2.34159.75688

[38] Youvan, D. C. (2025, August). Structural Cognition: Human and Artificial Intelligence in a Category Theory and Homotopy Type Theory World. ResearchGate preprint. DOI: 10.13140/RG.2.2.19060.26246

[39] Youvan, D. C. (2025, August). [Title not provided]. ResearchGate preprint. DOI: 10.13140/RG.2.2.32504.97286


rep
1
A
Hom
X
ϕ_X
A
RA